This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNS.2019.2912323, IEEE Transactions on Nuclear Science

Non-intrusive automatic compiler-guided reliability improvement of embedded applications under proton irradiation

Alejandro Serrano-Cases[1], Yolanda Morilla[2], Pedro Martín-Holgado[2], Sergio Cuenca-Asensi[1], and Antonio Martínez-Álvarez[1]

***Abstract*—A method is presented for automated improvement of embedded application reliability. The compilation process is guided using Genetic Algorithms and a Multi-Objective Optimization Approach (MOOGA). Even though modern compilers are not designed to generate reliable builds, they can be tuned to obtain compilations that improve their reliability, through simultaneous optimization of their fault coverage, execution time, and memory size. Experiments show that relevant reliability improvements can be obtained from efficient exploration of the compilation solutions space. Fault-injection simulation campaigns are performed to assess our proposal against different benchmarks and the results are assessed against a real ARM-based System on Chip under proton irradiation.**

***Index Terms*—Fault tolerance, single event upset, proton irradiation effects, soft errors**

I. Introduction

The use of Commercial-Off-The-Shelf (COTS) devices is under study as a serious competitor to the RadHard processors present in most critical scenarios, such as aerospace, control and safety systems, where radiation effects cause serious problems. Some of the reasons for the emergence of COTS are their low development costs, in part derived from reusing well-known design tools, thereby lowering non-redundant engineering costs, as well as their low power consumption and high computational power. However, the reduction in noise margins, due to the progressive reduction in technological resources and power operating levels, has the drawback of making these systems more susceptible to transient faults. Furthermore, common COTS processors are not designed to cope with those harmful effects of radiation and, because of their nature, traditional hardware redundancy techniques cannot be applied to their structural components.

In this context, Software-Implemented Hardware Fault Tolerance (SIHFT) techniques are intended to run reliable software over unreliable hardware [1]. Although these techniques increase reliability, the necessary instrumentation of their code causes important overheads in both memory footprints and execution times that deserve serious consideration [2], [3].

A potential method for modifying reliability without code instrumentation is by reproducing the way that modern compilers build the programs. In fact, if compiler parameters and flags are properly used, code can be reordered, useless instructions removed, unnecessary loops reduced, and constant operations precalculated, among many other optimizations. These changes produce different executables with the same functionality and may affect the observed reliability of the application. As a result, the same source code can be used to invoke many different executables with particular features, such as an improved execution time, a reduced memory footprint, and even increased fault coverage. In summary, SIHFT techniques gain reliability by instrumenting the code under protection, while compilers reorganize and optimize the code and, as a side effect, may improve its reliability.

In this context, modern compilers, such as *GNU-GCC* and *Clang/LLVM*, are known to offer a wide range of optimization parameters that are intended to reduce the code size or the execution time needed to complete the whole program. For instance, *Clang/LLVM* supports more than 250 optimizations and *GNU-GCC* offers 230 optimizations and 195 parameters for modifying those optimizations [4]. However, those compilers offer no predefined optimization associated with reliability improvements. Several studies in this area have approached the question of what influence the standard optimization levels have on application reliability. In [5], the authors analyzed how the first three optimization levels of *GCC* (named O1, O2 and O3) impacted on the expected number of failures in some specific processor structures. Medeiros et al. [6] added a further predefined optimization (Os) to their study and estimated the soft error resilience of 24 applications running on a SystemC model of an MIPS processor. Even though the results suggested that this flag provided better overall system behavior, in general, no clear relation was established between the standard optimizations, applications, and reliability enhancements. A similar conclusion can be found in [7], a study that concerned an ARM processor and emulated fault injection on the real hardware. When compared with the cross-sections obtained in heavy-ion experiments, the results showed different trends. According to the authors, that divergence could be explained by the partial injection campaigns, which targeted only the register file of the processor.

The overall picture becomes more complicated when considering all the available parameters and options. Iterative compilations [8] have shown important performance improvements that could be applied to reliability optimization. However, the computational effort that is required, similar to a brute force approach, makes any comprehensive exploration

[1]Dept. of Computer Technology, Ctra. San Vicente del Raspeig s/n, 03690, San Vicente del Raspeig - Alicante, Spain. Corresponding author: Antonio Martínez-Álvarez (email: amartinez@dtic.ua.es)

[2]Centro Nacional de Aceleradores (Universidad de Sevilla, CSIC, JA). Avda. Tomás Alba Edison 7, 41092 Sevilla, Spain

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNS.2019.2912323, IEEE Transactions on Nuclear Science

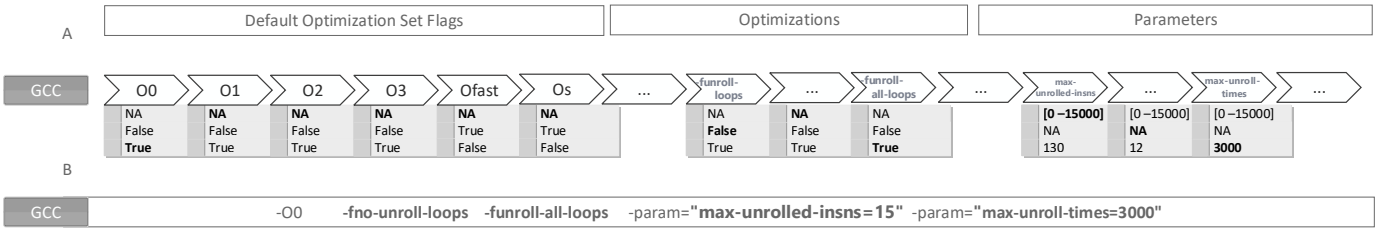

Fig. 1: A) Chromosome codification B) Example of a particular solution (individual).

of the solution space or even of a reduced subset of that space unfeasible. Narayanamurthy et al. [9] proposed the use of a Genetic Algorithm to alleviate that problem, which could identify compiler optimization sequences capable of improving application performance levels without degrading error resilience. The proposal was implemented without considering any specific processor (faults were injected on intermediate code) and the study was limited to a reduced subset of 10 optimizations provided by the *Clang/LLVM* compiler. A preliminary work of our own is presented in [10]. It combined *GA* with a Multi-Objective Optimization algorithm to explore the complete *GCC* solution space. The study presented a strategy based on register file vulnerabilities for improving the overall fault coverage of a particular low-end 16-bit processor.

Our above-mentioned work is continued and extended in this paper by studying a complex 32bit ARM-based architecture and *GCC*, one of the most widely used compilers. It presents the following contributions with respect to previous works. In the first place, and contrary to other approaches, our method takes into account all the *GCC* optimizations and parameters. As a result, it takes advantage of all the sophisticated processor features (out-of-order execution, branch predictors, pipeline, etc...) and conducts an in-depth exploration of compiler opportunities for the improvement of system reliability. In second place, in addition to the standard reliability factors -fault coverage and execution time- a new one is considered: memory size. As a consequence, our method offers new trade-offs to fit the system constraints. Finally, memory section vulnerability concurrently with the vulnerability of the register file is taken into account for estimating the fault coverage of the application, which represents a remarkable difference with respect to other approaches, because it increases the accuracy of the estimations. Furthermore, the solutions offered by our method, in a majority of cases, showed similar trends in proton-irradiation validation tests.

The search for the best compiler string that will in general serve to improve application reliability is a very complex task that lies beyond the scope of this work, due the large number of compiler optimizations under consideration. The main goal of our work is therefore to provide a method that, with any given application, will produce the executables with the best trade-offs among the three objectives that define its overall reliability.

Compared to traditional SIHFT techniques, which improve fault tolerance at the expense of important time and memory overheads, our method is simultaneously capable of increasing fault coverage and improving both performance and the memory footprint. The results show that, even when applying aggressive optimizations, the new approach can maintain and even increase fault coverage, and will consequently produce reliability increments of up to 4.2× in terms of the Mean Work to Failure metric. Our method is not designed to replace traditional redundancy techniques, but to complement and to reinforce them. In this way, compilation can be tuned before and after applying any specific SIHFT technique.

The rest of the paper will be organized as follows. Section 2 will start with a review of the compilation process and the role of the optimizations. It will then present our approach for tuning compilations. In section 3, the case study will be described together with the framework that is implemented to perform the space exploration. Similarly, in section 4, the details of the experimental setup used in radiation tests will be presented. Section 5 will show the solutions obtained by the *MOOGA* approach and, in a second subsection, those solutions will be compared with the radiation results. Finally, the conclusions of the work will be outlined in Section 6.

II. Compiler-guided reliability improvement using MOOGA

A. Background on Compiler Optimizations

Compilers evolved from simple source code translators of high-level code to machine code some time ago. The complexity of the first stages, known as front-end, has developed to the point where optimizations may even pass by undetected by programmers. While the last stages, known as back-end, have planning and resource utilization capabilities that can follow different strategies. These behaviors are controlled by users with a list of optimizations and parameters that are built into each compiler. It is a practical impossibility to establish the behavior of each one, due to their high number, for which reason compilers will usually offer a set of well-known optimizations. In the case of *GCC*, these optimizations, known as -O flags, will produce different levels of optimization. For instance, when -O1 is enabled, the compiler attempts to reduce execution time and code size, without performing any optimizations that will consume compilation time. -O2 compared to -O1, increases both, compilation time and performance, and -O3 introduces further optimizations. In turn, -O2 will enable code reorganization and will analyze the program to identify constant function parameters. While, -O3 introduces function inlining and removes loops with a relative low number of iterations, -O0 applies optimizations with a relatively low impact on the final executable, and -Os performs optimizations designed to reduce code size. Finally,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNS.2019.2912323, IEEE Transactions on Nuclear Science

-Ofast enables aggressive optimizations that could in some cases imply loss of accuracy. All these predefined optimization steps are centered on performance and memory footprint. They take no account of the reliability of the final application. Apart from the -O flags, *GCC* offers a lot of optimization steps, which usually have associated parameters for functional controls and can produce different effects. Some of them are intended to produce function cloning, to make inter-procedural constant propagation stronger (fipa-cp-clone), or they are designed to minimize stack usage (fconserve-stack), and others may have structural effects, such as parallelization and inline functions (finline-functions-called-once), or they may affect the instructions scheduling such as l2-cache-size and conserve-stack parameters. Some of them are bivaluated while others accept integer values.

In general, the effect of a combination of optimizations/parameters is difficult to predict. Even some flags and ranged parameters could have the same behavior depending on the problem. For instance, loop-unroll optimizations are used to speed up the calculation, reducing the number of jumps and variable checks. Depending on the unroll factor that is chosen, many different constructions can be generated with different size-performance-reliability trade-offs. However, if the unrolling factor is increased beyond the total number of iterations, no further effect will be produced, and no different build will be generated. Furthermore, if the unrolling factor applied is not a divisor of the number of iterations, the remaining iterations must be performed outside the main loop. Compared to the perfect unroll, the additional code reduces performance, increases the lifetime of some variables and, in some cases, may include new variable checks (e.g. when the number of iterations is unknown before compilation).

B. MOOGA approach

The potentially unworkable number of optimizations mentioned in the preceding section requires a strategy to accelerate the search for the combinations with the best features. In this context, we propose the use of Genetic Algorithms (*GAs*) [11] for efficient exploration of the solutions space, together with a Multi-Objective Optimization (*MOO*) algorithm[12] to deal with different objectives that affect the reliability of applications. The so-called *MOOGA* [13] approach will produce those candidates (individuals) that offer better trade-offs between each other.

GAs are probabilistic search algorithms used for high-dimensional stochastic problems with non-linear solutions. These groups of techniques define a branch of Evolutionary Algorithms (*EAs*) [14]. *GAs* are algorithms inspired by the evolution of the species. Thus, the individuals with better qualities have better chances of passing their genes to the next generation, while the worse are less likely to do so. *GAs* make use of the concept of crossover, to combine two individuals in a new one that shares the genes of both parents. There is also the concept of mutation, which randomly changes one gene from the genome of the individual. Crossover and mutation give *GA* the ability to perform a gradient descent search, with no stacking at local *minima*.

In our case, an individual is defined by a certain combination of compiler optimizations and parameters that become its genes (see Figure 1). In that way, individuals are coded using an array (chromosome) containing the state of each possible optimization parameter and flag. Each individual therefore describes a program compilation the behavior of which may differ from the other individuals.

In real-life problems, objectives that are under evaluation are not always independent from each other. The objectives are commonly related or in conflict with each other, which prevents simultaneous improvements. In such cases, *MOO* algorithms, which compute the weight of each objective function separately and then combine them in a single composite function, obtain the best compromise from among various objective functions.

In *MOO* sorting algorithms, the solutions will be ordered by the degree to which they meet the different objectives, so that the solutions reported by MOO are based on the concept of non-dominance of Pareto Efficiency. The Pareto Optimal Front shows multiples solutions with different degrees of satisfaction of the objectives. In addition, those solutions are characterized by their inability to improve any objective without worsening the others. Our approach makes use of the well-known NSGA-II [15] *MOO*. NSGA-II is a *MOO* algorithm based on non-dominant classifications, which constructs an initial arrangement based on non-dominated individual fronts. After the fronts are built, NSGA-II generates the individuals that belong to the same front in another order. NSGA-II uses the crowding distance function to estimate the diversity value of a solution. In that way, individuals are evaluated on the basis of their diversity within the dimension of each objective. The goal is to maintain a good spread of individuals and to increase the scope of the solution space that is explored.

In our case, the final executable needs to achieve improvements in fault coverage, performance and the memory footprint, which directly influence the overall reliability of an application. Those objectives and their inter-dependencies in embedded processors imply that the problem to be undertaken is a complex one.

MOOGA combines a Genetic Algorithm with Multi-Objective Optimization. Figure 2 shows how the combination of these two algorithms works to improve the overall reliability of an application. The first step is initialization, which oversees the gene encoding and produces a population of randomly generated individuals. The evolutionary loop is the second step, where our *MOOGA* approach iterates over several generations. Each generation is produced from the previous one crossing and mutating the best fitted individuals. Those individuals are evaluated by means of fault injection campaigns and ranked by *MOO* in terms of Pareto Efficiency. The process ends when a reliability goal is fulfilled or when a predefined number of iterations is reached. As a result, *MOOGA* processes all individuals on the Pareto Front, which were collected across the successive generations. Engineers can take advantage of all this information, to select the individual that best fits the system requirements. In this work, we selected some of them to be irradiated and to show the quality of the results that our proposal can offer.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNS.2019.2912323, IEEE Transactions on Nuclear Science

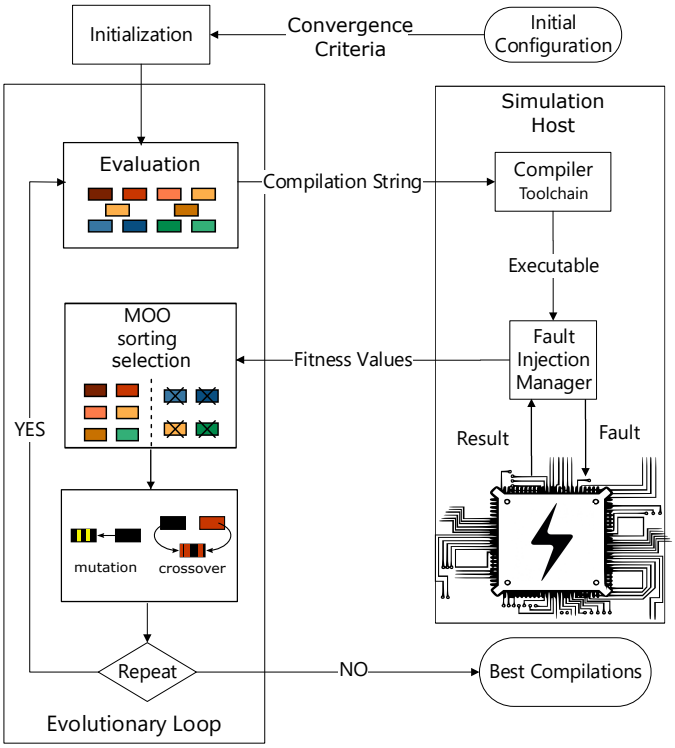

Fig. 2: *MOOGA* flowchart.

III. Simulation Setup

In an assessment of the strategy explained above, a set of benchmarks were selected from the Beebs (*Open Benchmarks for Energy Measurements on Embedded Platforms*) [16] project: *QuickSort, NDES, Dijkstra and BubbleSort*. *BubbleSort* is a sorting algorithm that involves basic loop constructs, integer comparisons, and simple array handling. *NDES* is a block-cipher based on a deterministic algorithm that operates on matrices stored in the memory known as keys, with between 65 and 640 elements. The algorithm takes a fixed-length block of 64bits and transforms it through different operations (permutations, substitution, xor, etc...) into another bitstring of the same length. The algorithm includes nested loops and deterministic memory access patterns. Additionally, keys integrity is crucial, because any minimal change in them could lead to the destruction of the data sets that are ciphered. Dijkstra is an algorithm that establishes the shortest path between nodes in a graph. The algorithm analyzes an adjacency matrix, which stores the weight of each route, following a random-access pattern. Finally, *QuickSort* is another sorting algorithm which operates in-place, requiring small additional amounts of memory to perform the task. The algorithms that were selected presented a variety of programming structures that are suitable for the application of different compiler optimizations. The compilations were performed by the *GCC* compiler from the Linaro project (version *arm-eabi-gcc v7.2-2017.11*).

A state-of-the-art ARM cortex A9 processor instruction-accurate model was the underlying architecture for each benchmarking test. It has a 32-bit CPU that includes a register file of 18 registers. The first 13 of them [R0-R12] are general purpose registers. The remainder, such as the stack pointer (SP), link register (LR), program counter (PC), Floating-point Status (FPS), and current program status register (CPSR) are control registers. The processor has a load/store architecture, which means that all the instructions operate with registers, except for load and store instructions. Cortex-A9 has a partial Out-of-order 8 stage pipeline that includes a branch prediction block and support for two levels of cache. Modern compilers, such as *GCC* that is used in this work, take advantage of all these sophisticated features to improve the executable code.

A. Fault Injector Manager

Once an application from the benchmark is compiled with a defined set of optimizations and parameters, its size in KiB of the corresponding executable was used as one of the objective functions. The second objective, Performance, was measured in terms of execution time using the Imperas OVPsim simulator [17], and was expressed in cycles. The evaluation of the fault coverage against soft errors was performed by means of fault injection campaigns, based on the bit-flip model with an injection of one fault per run. Each fault was emulated by means of a single bit-flip in a randomly selected bit from the resources (microprocessor register file and memory) and in a randomly selected clock cycle from the program duration. For this purpose, a custom plug-in was developed giving the simulator non-intrusive fault injection capabilities [18]. In doing so, no benchmarks were modified or instrumented with unnecessary code for injecting the faults. Moreover, this plug-in offers flexibility in the selection of the resources and the memory sections for fault injection. The boot code used to initialize the device was not considered in the injection, which yields fault coverage estimations of greater accuracy. An extension of Fault Injection Manager framework (*FIM*) [19] was used for conducting the Fault Injection Campaigns. *FIM* automatically gathers the ground truth parameters of an executable code, such as execution time and the memory map of the different sections. *FIM* controls the injection campaign by means of several user-defined parameters (e.g. number of faults, maximum allowable execution time, resources to be injected, etc.) and records the overall results. The fault effects are classified by *FIM* as *ACE - unnecessary for Architecturally Correct Execution*, in case the system completes its execution, and obtains the expected output after a fault is injected. Otherwise, they are classified as *ACE - Architecturally Correct Execution*, which comprises any undesirable effect categories such as uncorrected faults (SDC - *Silent Data Corruption*), abnormal program termination or infinite execution loop (*HANG*) [20]. Each campaign was configured to inject 1,000 faults per register in the register file and 18,000 faults in the memory segment allocated by the benchmark. This arrangement implies a total of 72,000 faults per individual (program version), achieving a statistical error of ± 0.01 at a 99% confidence level, according to the statistical model proposed by Leveugle et. al [21].

B. MOOGA Parameters

The *MOOGA* algorithm was configured to produce successive generations, each of 500 individuals. Our approach implemented the uniform mutation operator, and the probability

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNS.2019.2912323, IEEE Transactions on Nuclear Science

of change was therefore the same for each gene. Likewise, the uniform crossover operator was implemented, which is defined as the probability of exchanging each gene of the chromosome with some of its two parents. Mutation and crossover GA operators were set with a probability of 5% for most of the process. During the first phase of *MOOGA*, a high rate of mutated individuals was used to improve the *MOOGA* dynamic, assuring a richer population and accelerating the convergence of the algorithm.

The individuals that represent the main compilation flags of the *GCC* compiler were added to the initial population. These flags, as previously mentioned, are referred to as -O0, -O1, -O2, -O3, -Ofast, and -Os. They describe sets of well-known and reliable optimization strategies for: increasing performance (options -O0 up to -Ofast) and program size shrinking (-Os). The same flags were also used as reference points to compare the best individuals generated by *MOOGA*. Three objectives were selected for simultaneous optimization, because they are known to have a direct influence on program reliability: 1) memory footprint of the executable code, which defines the vulnerability area of the program; 2) execution time, which is proportional to the time that resources are exposed to faults; and, 3) the intrinsic vulnerability factor of the code expressed as the percentage of *ACE* faults. The simultaneous minimization of each objective defines our search space.

The metric Mean Work to Failure (*MWTF*) was also employed in this study. It was first defined by Reis et al. in [22] as the relation between the amount of work completed and the number of errors encountered. MWTF was designed to compare the effectiveness of different hardware and software techniques, as it captures the inherent trade-off between fault coverage improvements and the performance degradation that they produce. We likewise used this metric to compare the quality of different solutions obtained by our method. It is expressed as follows:

$$MWTF = \frac{1}{raw\_error\_rate \cdot AVF \cdot exec\_time} \quad (1)$$

where, the raw_error_rate is determined by the circuit technology. In our experiments, we used different executables running on the same device (technology), so this term of the equation can be considered as a constant and was not expressed in the results. The execution_time term is the time to execute a given unit of work. A unit of work is an abstract concept the specific definition of which depends on the application. In our case, work may be better defined as the execution of a program. *AVF* stands for Architecture Vulnerability Factor and is estimated by statistical fault injection and expressed as the *ACE* percentage. We used the convergence of the *MWTF* among those individuals belonging to the Pareto front and a maximum computational effort of 250 generations as the stop criterion for the *MOOGA* loop during the experimental tests. This number of generations was observed to be sufficient for the convergence of all the benchmarks.

IV. Radiation Setup

The Device Under Test (*DUT*) selected for the irradiation experiment was the ZYBO board, equipped with a 28nm CMOS *Xilinx ZYNQ XC7Z010* System On Chip (SoC). This SoC is divided into two parts, an FPGA area (Programmable Logic – PL) and a 32-bit ARM Cortex A9 microprocessor (Processing System – PS). In addition, the microprocessor has a built-in memory called On Chip Memory (OCM), onto which the bootloader or the test program can be loaded. The test application was compiled with the same compiler as in the simulation, adding the Board Support Package (BSP) provided by Xilinx to initialize the *DUT*.

The *DUT* was controlled by an external computer, the *RaspberryPi 3 Model B*, the main task of which was to receive and log all the messages sent by the *DUT*. The *DUT* was configured to send a state message every 5s in the absence of errors, otherwise the message would be instantly notified and the external computer would reset and reprogram the *DUT*.

The test campaign was performed at the National Centre for Accelerators, in Spain, at the start of 2018 [23]. The irradiation tests were performed using the external beam line, installed in the cyclotron laboratory. Although the proton energy delivered by this cyclotron was set at 18 MeV, the beam extraction system was upwards towards the air gap to irradiate the *DUT*. In this case, the *DUT* was placed at 53.5 cm from the exit nozzle with a mylar foil window of $125\mu m$, so that the final energy at the surface was 15.2 MeV, with an estimated spread of $\sim$300 KeV. Previous tests at the CNA have shown that the energy range of incident protons in the silicon active area, 10 to 8 MeV, is sufficient to produce events without thinning them [24]. The final energy of the incident beam at the surface and in the active area was obtained by using the energy loss data calculated with the SRIM2013 code [25].

Proton flux monitoring was performed indirectly, as the direct current reading on the *DUT* was not available. During the tests, the beam current was measured in an electrically isolated graphite collimator situated behind the exit window. In this study, a Brookhaven 1000c current integrator was used at a frequency scale of 600 pA (10 pA sensitivity). With daily calibrations, a correlation factor was achieved by simultaneous measurements into the graphite collimator and another graphite plate at the *DUT* position. In addition, a grounded aluminum mask in front of the target was used to avoid induced currents effects between both items and to define a uniform area of irradiation.

The flux value was constant and fluctuated under 5% during each run. A medium flux value was calculated, based on the pulses registered by the counter. Finally, the fluence at the device under test was calculated as a function of the exposure time for each run with an accuracy of 10%. Under these experimental conditions, the beam uniformity was higher than 90% in the area of interest.

V. Experimental results and discussion

Prior to the irradiation, a *MOOGA* optimization stage was performed on the entire benchmark suite considering all *GCC* options and parameters. The computational effort required for each application differed in accordance with its complexity and the number of generations needed for *MOOGA* convergence. The most computationally intensive application

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNS.2019.2912323, IEEE Transactions on Nuclear Science

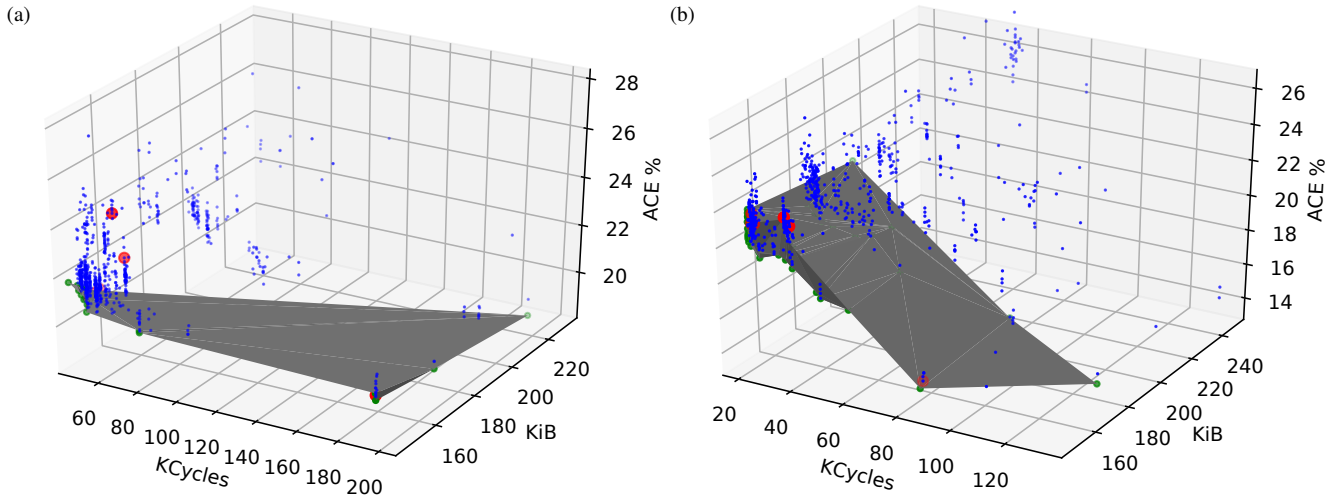

Fig. 3: 3D view of *BubbleSort* (a) and *NDES* (b) individuals generated with the *MOOGA* technique: singular individuals are shown in red and the Pareto surface in gray.

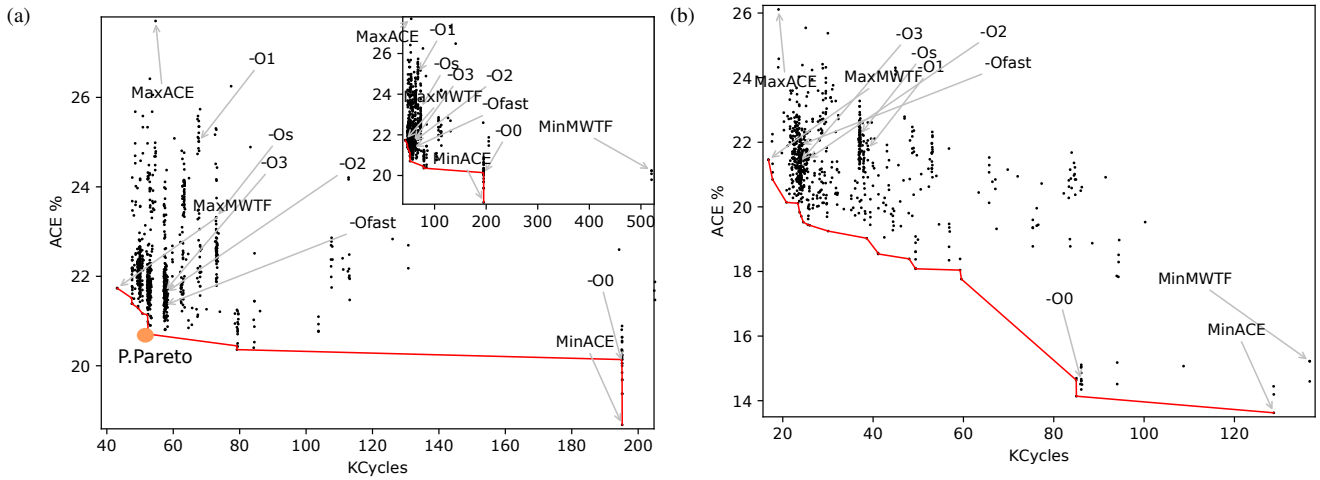

Fig. 4: Fault coverage (*ACE%*) against execution time (Kcycles) for *BubbleSort* (a) and *NDES* (b) individuals generated with the *MOOGA* technique. Singular individuals are labelled and the Pareto front is highlighted in red.

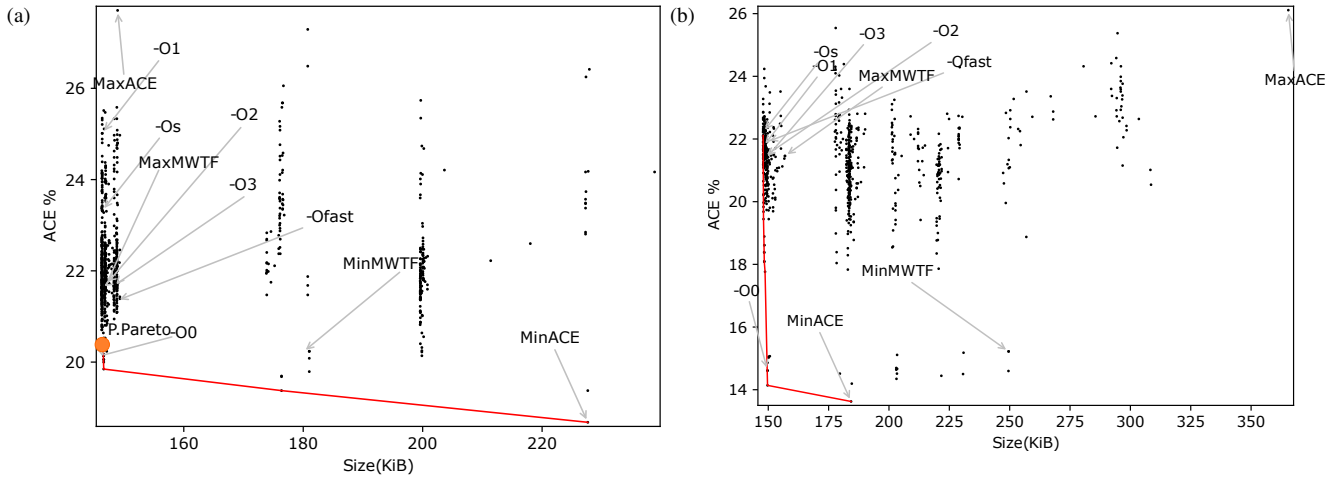

Fig. 5: Fault coverage (*ACE%*) against the memory footprint (KiB) for *BubbleSort* (a) and *NDES* (b) individuals generated with the *MOOGA* technique. Singular individuals are labelled and the Pareto front is highlighted in red.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNS.2019.2912323, IEEE Transactions on Nuclear Science

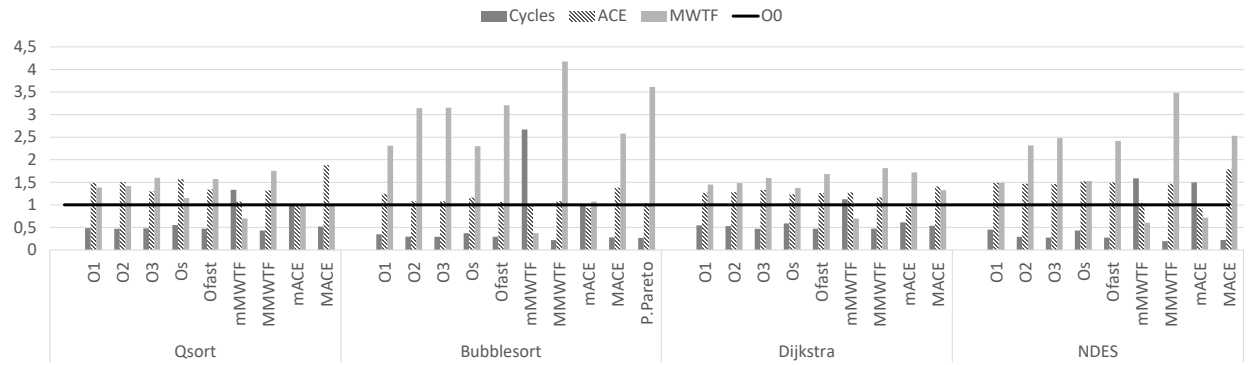

Fig. 6: Simulation results normalized to *O0* (line) for all the relevant individuals. Lower values indicate better performance (Cycles) and fault coverage (*ACE* rates), while higher values indicate better *MWTF*

was *NDES*. In that case, a single fault injection campaign (72,000 faults) lasted between 3 and 7 minutes depending on the individual. The *MOOGA* implementation was improved to skip the evaluation of equivalent individuals (i.e. their chromosomes included different optimizations and parameters but they produced identical executable files). In that way, the whole tuning process of *NDES* lasted 5 days, running on a PC desktop with an x86 processor (Intel core i5).

A. MOOGA simulation

For the sake of simplicity, only two applications are shown in Figures 3, 4 and 5, where the algorithms *BubbleSort* and *NDES* are represented in subplots a and b, respectively.

Figure 3 shows the population obtained from the *MOOGA* optimization process and the distribution of the individuals (small size blue dots) around the solution space. The features of the algorithm, such as instruction level parallelism, memory access patterns, data and control structures, among others, influence the capabilities of the compiler for the effective application of its entire arsenal of optimizations. *BubbleSort* is one of the simplest sorting algorithms, which presents low instruction parallelism and interacts poorly with modern processor hardware. It produces at least twice as many writes as other more sophisticated sorting algorithms (e.g. insertion sort), twice as many cache misses, and more branch mispredictions. It also translates into very limited number of optimizations with an observable effect in the executable file. Figure 3.a shows the reduced solution space of *BubbleSort*, which includes several groups of nearly identical individuals. On the contrary, the *NDES* algorithm presented a higher level of instruction parallelism, deterministic loops and repetitive access patterns. All these features permit a deeper exploration of the optimization space, as can be seen in Figure 3.b that show greater differentiation of individuals and a wider population spread. In both cases, *BubbleSort* and *NDES*, the individuals are grouped in clusters, which indicates the presence of optimization sets that have a strong incidence on the objectives under evaluation. Another important element is the Pareto surface (in gray), which represents the frontier of enhancements in which the best candidates are represented (medium size green dots). Likewise, singular individuals are shown with large size red dots.

For a detailed analysis of the population, the individuals are shown in figures 4 and 5 facing pairs of objectives. The *ACE* rate against the execution time is shown in Figure 4. In both applications, the fault coverage range of the population is significant. For instance, fault coverage of the *BubbleSort* populations range between 27.7%*ACE* (*MaxACE*) and 18.6% *ACE* (*MinACE*), while the number of cycles needed to complete the programs varies from its minimum located at 43 Kcycles, up to its maximum located at 520 Kcycles. Looking at the third objective under evaluation (Figure 5), it can be appreciated that it ranges from 238 KiB to 146 KiB. In summary, the fault coverage can be improved by nearly 9.1%, while its performance can be improved by about 12× and the memory overhead reduced by 1.6×, simply by tuning the compilation process. The *NDES* algorithm showed similar behavior to *BubbleSort*. In this case, the difference was nearly 12% in the *ACE* rate between *MaxACE* and *MinACE* individuals, the performance variation was close to 5× and the memory overhead was about 2.5×.

Figures 4 and 5 also reveal the complex relations between the objectives. Intuitively, larger memory usage will produce applications that are more prone to faults. However, this relation is unclear and it depends on other factors. For instance, if the memory footprint increases, but the lifetime of the stored variables decreases, it could lead to a fault coverage improvement. This effect can be seen in Figure 5.a, where individuals with a similar memory footprint (about 200KiB) present a range of *ACE* percentages between 20% and 26%. Moreover, the individual with the minimum *ACE* (i.e. maximum fault coverage) presents a maximum memory footprint of around 230KiB. Figure 5.b shows higher differences in the same case of 200KiB, where *ACE* varies between 14% and 25%. Therefore, the key is not the memory size but the way the memory is used. This fact corroborates the relevance of considering memory size as the third optimization objective in our approach. An analogous behavior can be observed in Figure 4, where the individuals that present very close execution times form vertical clusters with a significant variability in fault coverage. Indeed, the relationship between these three parameters is not clear, and it is not possible *a priori* to establish how they will evolve. These are the trade-offs that our algorithm is seeking to exploit.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNS.2019.2912323, IEEE Transactions on Nuclear Science

The behavior of the main optimization flags can be observed in Figure 4.a. The individual *O0* has the best fault coverage (20% *ACE*), at the cost of having 2× more cycles than the others. Focusing now on the best individuals in terms of performance, degradation can be seen in their fault coverage. For instance, *O1* from Figure 4.a, which has the lowest fault coverage, lower than 5% *O0*. Likewise, the higher optimization levels -*O2, O3, Ofast, Os*- showed similar behaviors to *O1* in terms of performance, while in terms of fault coverage, this group lowered its reliability by around 2%. Regarding *NDES* (Figure 4.b), the set *O1, O2, O3, Ofast, Os* showed an increase in the *ACE* rate of 8% compared to the worst option (*O0*). This significant worsening in fault coverage was at the expense of a performance increase of ∼4×.

Prior to the radiation experiment, some relevant individuals were characterized. Figure 6 presents a summary of the improvements to the objectives for the main optimization flags and for the individuals with *Maximum* and *Minimum ACE* percentage and those with *Maximum* and *Minimum MWTF* metrics. The fault coverage variations between them is remarkable, if we consider the individual *O0* (baseline), which is the reference compilation with no optimization at all from among all the benchmarks under evaluation. It can be seen that this build had the best fault coverage, except for the *minACE* build, at the expense of more cycles than the other individuals.

As can be seen, *MOOGA* can obtain individuals with a similar fault coverage to *O0*, but much shorter execution times, resulting in important improvements of *MWTF* (e.g. *BubbleSort* individual *MaxMWTF* improved this metric by as much as 4×). A significant individual in the *BubbleSort* experimental test was the one labeled *P.Pareto*, located in the corner of the Pareto frontier, which offered a good trade-off between the objectives under evaluation (Figure 4.a and 5.a).

Regarding the remaining optimizations, the cycle speedup led to a worsening of the *ACE* results. As the final objective was to reduce this *ACE* rate to a *minimum*, in so far as possible with no loss of previously acquired speedup, the only builds to achieve those objectives were the ones that applied the *MWTF* metric. This metric not only takes into account the *ACE* rate of the program, but also the time needed to complete it. As can be seen from the results of all the programs, *MinMWTF* individuals were characterized by having the highest number of cycles and a relatively low *ACE* rate. On the contrary, *MaxMWTF* individuals on the Pareto frontier were the best in terms of performance.

fsched-stalled-insns=

Finally, it is worth comparing those numbers with the ones obtained in our previous study on a simpler 16-bit processor (TI-MSP430). In that case, considering the worst and the best individuals of the *MOOGA* approach, it achieved improvements of up to 6% in fault coverage and up to 45% for the *MWTF* metric. Meanwhile, this new study on ARM, considering the *O0* (not the worst individuals) as the baseline, showed improvements of up to 13% in fault coverage and of up to 420% for the *MWTF* metric. Even if the inaccuracies in the MSP executables evaluation (the memory was excluded from the injection campaigns) are considered, those remarkable differences reveal that sophisticated ARM architectural features, such as out-of-order execution, speculative execution, instruction pipelines and branch predictions, play an important role in the reliability improvements, permitting the compiler to squeeze the high-level optimizations. This hypothesis is corroborated when comparing the number of different individuals (unique individuals) produced by our method for MSP and ARM. A maximum of 51 unique individuals were obtained (Synthetic program) for MSP, while *MOOGA* explored more than 722 unique individuals (*NDES* program) for ARM.

B. Radiation stage

The candidates were finally reduced to the sorting (*BubbleSort*) and the cipher (*NDES*) algorithms, due to beam limitations. The individuals chosen for irradiation were: *MaxACE*, O0, O3, *MaxMWTF*, *MinMWTF* and *P.Pareto* for *BubbleSort*; and, *MinACE*, *MaxMWTF*, O0 for *NDES*.

The irradiation results are presented in Table I, which shows the SDC and the HANG dynamic cross-sections, the MWTF values, and both the fluxes and the fluences for each individual. The aforementioned candidates were customized to use either DDR (out of the incident proton beam) or on-chip OCM memory. The 95% confidence intervals are included in all cases. These confidence intervals are computed using the classical formula for the estimation of the Poisson mean [26].

The OCM radiation setup, i.e. when all resources (memories and registers) are inside the beam, defines the most similar scenario to our MOOGA simulation. In this case, and looking at the *BubbleSort* section of the table, a remarkable match between the MOOGA estimations and the radiation results can be appreciated in the following terms. Firstly, the individuals labelled by MOOGA as *MaxACE* and *MinMWTF* i.e. the individuals with the worst fault coverage, obtained higher dynamic cross-sections ($3.8 \cdot 10^{-11}$ and $4.8 \cdot 10^{-11}$, respectively). A minimum discrepancy can be seen where the *MaxACE* individual was supposed to have the highest dynamic cross-section, but this was obtained by the *MinMWTF* version. One possible explanation is that additional cycles will have a decisive impact on fault coverage. The resources omitted during the fault injection campaigns (e.g. pipeline registers) could likewise lead to the same result, as suggested by other authors [7]. Secondly, the individuals showing the best fault coverage estimations (*O0*, *MaxMWTF* and *P.Pareto*), as expected, presented a lower dynamic cross-section than the others. Thirdly, the *MaxMWTF* and *P.Pareto* individuals showed higher values of *MWTF* under radiation ($4.67 \cdot 10^{14}$ and $5.45 \cdot 10^{14}$, respectively). The *P.Pareto* version improved its execution time compared with the simulation. As a result, the *P.Pareto* version recorded better *MWTF* metrics in the irradiation experiment.

Similarly, in the *NDES* OCM section of the table, it can be seen that the *MinACE* and the *O0* individuals, as expected, produced lower dynamic cross-sections ($5.9 \cdot 10^{-11}$ and $5.2 \cdot 10^{-11}$ respectively) than the others. Also, *MaxMWTF* produced the maximum *MWTF* under radiation ($5.47 \cdot 10^{14}$).

Regarding the cache-deactivated DDR radiation setup, where memory errors are minimized (out of the beam), the following effects were observed in the *BubbleSort* benchmark.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNS.2019.2912323, IEEE Transactions on Nuclear Science

		Version	cache	Flux $p/cm^2 \cdot s$	Fluence p/cm^2	Cycles	SDC	Hang	σ_{SDC} $(10^{-11})cm^2$	σ_{Hang} $(10^{-11})cm^2$	σ_{Total} $(10^{-11})cm^2$	MWTF
BubbleSort	OCM	MaxACE	✓	$7.8 \cdot 10^8$	$3.4 \cdot 10^{12}$	44409	112	14	3.3(2.7, 3.9)	0.42(0.20, 0.64)	3.8(3.2, 4.4)	$3.96 \cdot 10^{14}$
		-O0	✓	$7.4 \cdot 10^8$	$4.9 \cdot 10^{12}$	220842	76	40	1.6(1.3, 1.9)	0.82(0.57, 1.1)	2.4(2.0, 2.8)	$1.26 \cdot 10^{14}$
		MaxMWTF	✓	$8.3 \cdot 10^8$	$3.1 \cdot 10^{12}$	38412	79	36	2.5(1.9, 3.1)	1.2(0.82, 1.6)	3.7(3.0, 4.4)	$4.67 \cdot 10^{14}$
		MinMWTF	✓	$1.0 \cdot 10^9$	$2.4 \cdot 10^{12}$	389806	69	45	2.9(2.2, 3.6)	1.9(1.4, 2.4)	4.8(3.9, 5.7)	$3.54 \cdot 10^{13}$
		P.Pareto	✓	$1.1 \cdot 10^9$	$3.4 \cdot 10^{12}$	39635	65	38	1.9(1.4, 2.4)	1.1(0.74, 1.5)	3.1(2.5, 3.7)	$5.45 \cdot 10^{14}$
	DDR	-O3	✗	$1.1 \cdot 10^9$	$8.6 \cdot 10^{12}$	646595	15	44	0.17(0.08, 0.26)	0.51(0.36, 0.66)	0.68(0.50, 0.86)	$1.50 \cdot 10^{14}$
		MaxACE	✗	$2.5 \cdot 10^9$	$7.4 \cdot 10^{12}$	666005	32	68	0.43(0.28, 0.58)	0.91(0.69, 1.1)	1.3(1.0, 1.6)	$7.38 \cdot 10^{13}$
		MinMWTF	✗	$2.5 \cdot 10^9$	$1.0 \cdot 10^{13}$	7495395	14	88	0.14(0.07, 0.21)	0.85(0.67, 1.0)	0.98(0.78, 1.2)	$8.95 \cdot 10^{12}$
NDES	OCM	MinACE	✓	$1.2 \cdot 10^9$	$1.8 \cdot 10^{12}$	127210	72	35	4.0(3.1, 4.9)	1.9(1.3, 2.5)	5.9(4.8, 7.0)	$8.82 \cdot 10^{13}$
		MaxMWTF	✓	$1.1 \cdot 10^9$	$9.4 \cdot 10^{11}$	10658	91	15	9.7(7.7, 12)	1.6(0.79, 2.4)	11(8.9, 13)	$5.47 \cdot 10^{14}$
		-O0	✓	$9.7 \cdot 10^8$	$2.2 \cdot 10^{12}$	93639	80	33	3.7(2.9, 4.5)	1.5(0.99, 2.0)	5.2(4.3, 6.1)	$1.36 \cdot 10^{14}$
	DDR	-O0	✓	$9.9 \cdot 10^8$	$1.6 \cdot 10^{12}$	93954	82	29	5.1(4.0, 6.2)	1.8(1.1, 2.5)	6.9(5.6, 8.2)	$1.01 \cdot 10^{14}$

TABLE I: Summary of radiation results for each version of the selected applications. The flux uncertainty is $\pm 10\%$.

Firstly, the overall dynamic cross-section was, as expected, reduced with respect to the OCM version. For instance, the *MinMWTF* version fell from $4.8 \cdot 10^{-11}$ down to $9.8 \cdot 10^{-12}$ and from $3.8 \cdot 10^{-11}$ down to $1.3 \cdot 10^{-11}$ in the case of *MaxACE*. Secondly, the *MaxACE* also offered the maximum dynamic cross-section. Thirdly, the *MinMWTF* version produced the worst *MWTF* value, $8.95 \cdot 10^{-12}$, that was measured.

Finally, it is interesting to note that if the program fits in the cache, then the behavior of both the OCM and the DDR versions will be similar. This case occurred with *NDES O0*, where the dynamic cross-section varied slightly from $5.2 \cdot 10^{-11}$ in OCM up to $6.9 \cdot 10^{-11}$ in DDR. Additionally, the cache disablement exposed a performance worsening side effect of 15×. For instance, *BubbleSort* MaxACE rose from 44.4 Kcycles in OCM to 666.0 Kcycles in DDR. Consequently, the *MWTF* was worse compared with the cache-on individuals.

All the above examples provided evidence of a significant match between simulation and radiation results, and validated the MOOGA approach for the generation of relevant individuals.

The analysis is complemented by Figure 7. It shows fault distribution between SDC and Hang, comparing simulation and radiation results. Also, the number of memory accesses is represented by a line (referred to the right axis). Considering the OCM setup, a quick look at the *BubbleSort* individuals reveals that the SDC/Hang ratios are very similar in both cases (differences of less than 15%), regardless of the number of memory accesses. The *NDES MaxMWTF* individual presents a similar trend. However, for the *MinACE* and the *O0* individuals, which perform a higher number of memory accesses, the SDC/Hang ratio is reversed. A possible explanation of this effect is the fact that data involved in the *NDES* calculus are more sensitive than those involved in *BubbleSort*. Regarding the DDR setup, and conversely to simulation results, the Hang percentage is dominant regardless of the number of memory accesses (see *BubbleSort MaxACE* and *MinMWTF* versions). It shows that out-beam stored data are less vulnerable to SDC faults.

VI. Conclusion

It has been demonstrated in this study that reliability can be improved by tuning the compilation process. A blind automatic strategy has also been proposed to guide the search for the versions with the best trade-offs between several objectives which influence application reliability. Despite the fact that modern compilers are not designed to generate reliable builds, they can be tuned to generate compilations that improve their reliability by means of simultaneous optimization of the fault coverage, the execution time, and the memory size. Moreover, it can be inferred from comparisons with previous studies on a simpler processor that sophisticated hardware features play an important role in the reliability improvements that can be achieved through efficient optimizations of compilers.

Acknowledgment

This work was funded by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund through the following projects: '*Evaluación temprana de los efectos de radiación mediante simulación y virtualización. Estrategias de mitigación en arquitecturas de microprocesadores avanzados*' and '*Centro de Ensayos Combinados de Irradiación*', (Refs: `ESP2015-68245-C4-3-P` and `ESP2015-68245-C4-4-P`, MINECO/FEDER, UE).

References

[1] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-implemented hardware fault tolerance*. Boston, MA: Springer US, 2006. [Online]. Available: https://doi.org/10.1007/0-387-32937-4

[2] F. Restrepo-Calle, A. Martínez-Álvarez, S. Cuenca-Asensi, and A. Jimeno-Morenilla, "Selective SWIFT-R: A flexible software-based technique for soft error mitigation in low-cost embedded systems," *Journal of Electronic Testing: Theory and Applications (JETTA)*, vol. 29, no. 6, pp. 825–838, Dec 2013.

[3] A. Martínez-Álvarez, S. Cuenca-Asensi, F. Restrepo-Calle, F. R. Palomo Pinto, H. Guzman-Miranda, and M. A. Aguirre, "Compiler-directed soft error mitigation for embedded systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 2, pp. 159–172, March 2012.

[4] "GNU Compiler Collection: Optimize Options," 2018. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TNS.2019.2912323, IEEE Transactions on Nuclear Science

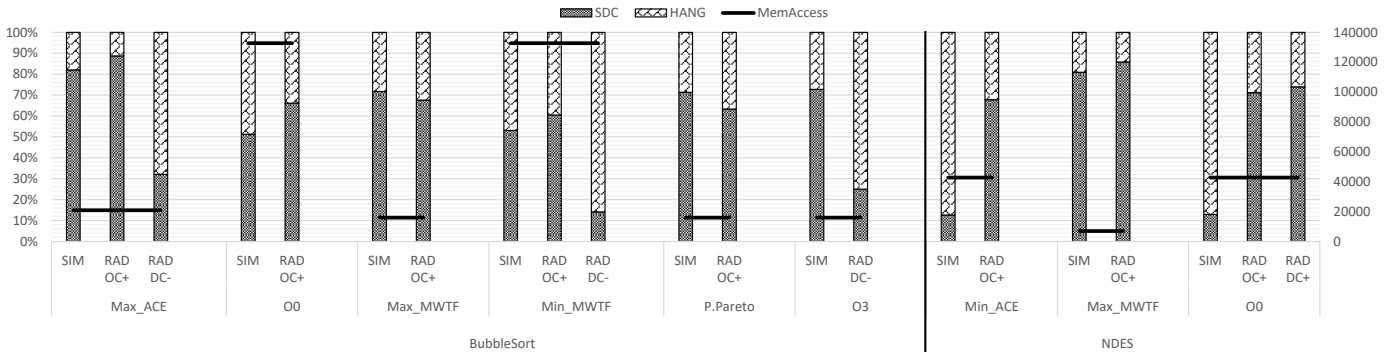

Fig. 7: Simulated (SIM) and radiated (RAD) *SDC* and *HANG* normalized. The Radiation results are classified depending on the resource where the program resides *OCM* (O) or *DDR* (D) and the activation (C+) or deactivation (C-) of the cache. The line represents the number of memory accesses executed by each build on simulation.

[5] M. Demertzi, M. Annavaram, and M. Hall, "Analyzing the effects of compiler optimizations on application reliability," in *Proceedings - 2011 IEEE International Symposium on Workload Characterization, IISWC - 2011*. IEEE, Nov 2011, pp. 184–193.

[6] L. O. Guilherme E. Medeiros, Felipe T. Bortolon, Ricardo Reis, "Evaluation of Compiler Optimization Flags Effects on Soft Error Resiliency," in *Symposium on Integrated Circuits and Systems Design (SBCCI)*. IEEE, Aug 2018, pp. 1–6.

[7] F. M. Lins, L. A. Tambara, F. L. Kastensmidt, and P. Rech, "Register File Criticality and Compiler Optimization Effects on Embedded Microprocessor Reliability," *IEEE Transactions on Nuclear Science*, vol. 64, no. 8, pp. 2179–2187, 2017.

[8] E. Park, S. Kulkarni, and J. Cavazos, "An evaluation of different modeling techniques for iterative compilation," in *2011 Proceedings of the 14th International Conference on Compilers Architectures and Synthesis for Embedded Systems CASES*. New York, New York, USA: ACM Press, 2011, pp. 65–74.

[9] N. Narayanamurthy, K. Pattabiraman, and M. Ripeanu, "Finding Resilience-Friendly Compiler Optimizations Using Meta-Heuristic Search Techniques," in *Proceedings - 2016 12th European Dependable Computing Conference, EDCC 2016*. IEEE, Sep 2016, pp. 1–12.

[10] A. Serrano-Cases, J. Isaza-Gonzalez, S. Cuenca-Asensi, and A. Martinez-Alvarez, "On the influence of compiler optimizations in the fault tolerance of embedded systems," in *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design, IOLTS 2016*. IEEE, Jul 2016, pp. 207–208.

[11] Kramer, Oliver, "Genetic Algorithms," in *Genetic Algorithm Essentials*. Springer International Publishing, 2017, pp. 11–19.

[12] O. Kramer, "Multiple Objectives," in *Genetic Algorithm Essentials*. Springer International Publishing, 2017, pp. 47–54.

[13] S. Bechikh, M. Elarbi, and L. Ben Said, "Many-objective optimization using evolutionary algorithms: A survey," in *Adaptation, Learning, and Optimization*. Springer International Publishing, 2017, vol. 20, pp. 105–137.

[14] S. B. Alain Petrowski, *Evolutionary Algorithms*, ser. Metaheuristics Set vol. 9. ISTE / Wiley, 2017.

[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr 2002.

[16] J. Pallister, S. J. Hollis, and J. Bennett, "BEEBS: open benchmarks for energy measurements on embedded platforms," *CoRR*, vol. abs/1308.5174, 2013.

[17] "OVPsim," 2018. [Online]. Available: www.ovpworld.org/technology_ovpsim

[18] P. Yuste, J. C. Ruiz, L. Lemus, and P. Gil, "Non-intrusive Software-Implemented Fault Injection in Embedded Systems," in *Dependable Computing*. Springer Berlin Heidelberg, 2003, pp. 23–38.

[19] J. Isaza-Gonzalez, A. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi, and A. Martinez-Alvarez, "Dependability evaluation of COTS microprocessors via on-chip debugging facilities," in *LATS 2016 - 17th IEEE Latin-American Test Symposium*, 2016, pp. 27–32.

[20] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2003, vol. 2003-Janua, pp. 29–40.

[21] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," *2009 Design, Automation & Test in Europe Conference & Exhibition*, pp. 502–506, 2009.

[22] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Design and evaluation of hybrid fault-detection systems," in *Proceedings - International Symposium on Computer Architecture*. IEEE, 2005, pp. 148–159.

[23] Centro Nacional de Acelaradores, "Seville, Spain," http://www.cna.us.es, Last visited: July 10th.

[24] A. Lindoso, M. Garcia-Valderas, L. Entrena, Y. Morilla, and P. Martin-Holgado, "Evaluation of the suitability of NEON SIMD microprocessor extensions under proton irradiation," *IEEE Transactions on Nuclear Science*, vol. 65, no. 8, pp. 1835–1842, Aug 2018.

[25] J. Z. et al., "SRIM The Stopping and Range of Ions in Matter. SRIM Co." http://www.srim.org, 2010.

[26] A. W. K. Norman L. Johnson, Samuel Kotz, *Univariate discrete distributions*, 2nd ed., ser. Wiley series in probability and mathematical statistics. Applied probability and statistics. Wiley, 1992.